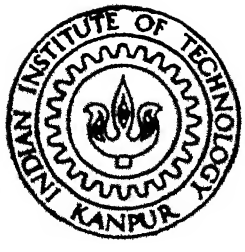
# Design and Implementation of Transparent Anonymous FTP for Linux

by

T. Kiran




CSE
1998
M
KIR
DES

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

**INDIAN INSTITUTE OF TECHNOLOGY KANPUR**

**MARCH, 1998**

# Design and Implementation of Transparent Anonymous FTP for Linux

*A Thesis Submitted*
*in Partial Fulfillment of the Requirements*
*for the Degree of*
*Master of Technology*

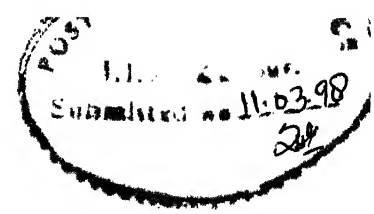
*by*
*T. Kiran*

*to the*
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
March, 1998

5 MAY 1998 / CSE
CENTRAL LIBRARY
I. I. T. KANPUR
Acc. No. A 125439

CSE-1998-M- KIR-DES

Entered in System
Nimisha
2.6.98



# Certificate

Certified that the work contained in the thesis entitled "*Design and Implementation of Transparent Anonymous FTP for Linux*", by Mr.*T. Kiran*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Dr. Deepak Gupta)
Assistant Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

March, 1998

## Abstract

This thesis describes the design and implementation of a system that allows the users to access files on remote anonymous FTP sites transparently. By transparency we mean that all files on all remote FTP sites in the world appear to be the part of the local file system tree and can be accessed using any of the familiar Unix programs without the need to modify or even recompile these programs. This is achieved by implementing a new type of file system, called the FTP file system, for Linux. The FTP file system implements the file transfer protocol (FTP) inside the kernel and makes files on remote archive sites appear as local files. It uses a disk cache to cache recently accessed files. A user level process, called the cache daemon, periodically deletes cached files that satisfy the system administrator specified criteria, in order to provide some semblance of cache coherence.

# Acknowledgments

I wish to express my sincere gratitude to Dr. Deepak Gupta for his invaluable guidance throughout the work. I also thank him for being my mentor and encouraging me in every possible way.

My sincere thanks to the lab staff in the Department of Computer Science for their assistance and cooperation during this work. I would like to specifically thank Sri Ram, Kishore, Madhu, and Yugandhar who have helped during the course of this thesis work. All my classmates have also made my stay here a most memorable one.

I am grateful to my parents and my brothers for being a constant source of motivation.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The basic functionality of a network is to provide transfer of data between computers. This functionality ensures that all the data on a computer connected to a network is accessible from any other computer in that network. Not only this, by adding data onto a computer, it is immediately available to all the users of the network. The Internet is a huge network of computer networks around the world that facilitates communication among all computers connected to these networks. Because of the large number of hosts in the Internet, it has a vast amount of information and this information is growing every day. The Internet also provides many services such as electronic mail, interactive conferences, network news and the ability to transfer files. The information in the Internet is made available to users through web pages and the archive sites. An archive site is a host in the Internet which acts as a repository of information. Information stored on these Internet hosts is made available for users to transfer to their local sites. Users run software to identify this information and transfer it to their own hosts. Such a transfer is done with a program that implements the File Transfer Protocol (FTP).

These programs only support basic features such as file transfer, directory listing etc. Sophisticated operations such as locating files on the server with a specified name are not possible with the usual client programs. On the other hand there are powerful utilities available on most systems for performing such sophisticated

operations on the local file system. These utilities however cannot work with the server file system.

This work aims to provide a logical view to the user in which all the files on archive sites seem to be part of the local file system hierarchy. By this, we mean that the user should be able to access files on archive sites as if they were on the local disk. This enables the existing software such as file managers and other utilities, which are useful in accessing the local file system, to be used to access archive sites without modifying or even recompiling them. In this way, many different types of interfaces i.e., from the Unix shell type interface to the graphical user interface, are immediately available to access files on the archive sites and programs to implement them already exist. A user may choose a familiar interface to access these archive sites. This relieves the user of the burden of learning the new interface provided by a program which implements the FTP. These interfaces also provide more functionality than a program which implements the FTP. For example, in a shell type interface, a user can check the existence of a file in an archive site by using the standard Unix command *find*.

## 1.1 Organization of the Report

The rest of this report is organized as follows. In Chapter 2, we briefly review the background concepts which are required to understand the design of our system. Essentially this consists of a few basics of the Linux file system and a brief description of the file transfer protocol. In Chapter 3, we present a detailed discussion of the system design and various issues that are related to the design. In Chapter 4, the implementation of the FTP file system is described. In Chapter 5, we present the design of a cache daemon program which is used to delete obsolete files from the cache. Finally in Chapter 6, we compare the performance of our system with that of other software to transfer files and present some concluding remarks.

# Chapter 2

# Background

In this chapter, we briefly describe the background concepts needed to understand this work i.e., the Linux file system and the file transfer protocol.

## 2.1 Linux File System

In this section we briefly describes the Linux file system architecture. The interested reader is referred to [2] for details. The main goal of a file system is to provide a uniform view of data on all forms of storage devices and the operations to access this data. By defining a file as a basic unit of abstraction for storage, the storage device can be viewed as a tree with a root node called *root*; each non-leaf node is a *directory* file and a leaf node is either a directory file, a regular file, or a device file. A regular file is a stream of data bytes and no meaning is associated with this data. The information required to manage these files is kept apart from the data and is collected in a structure called *inode*. The information in an inode includes access time, access rights and the allocation of data blocks on the physical media. Directories are special types of files whose data contains the file names and inode numbers of the files in the directory. Device files are special type of files that represent physical devices such as terminals and printers but they are accessed as if they were regular files.

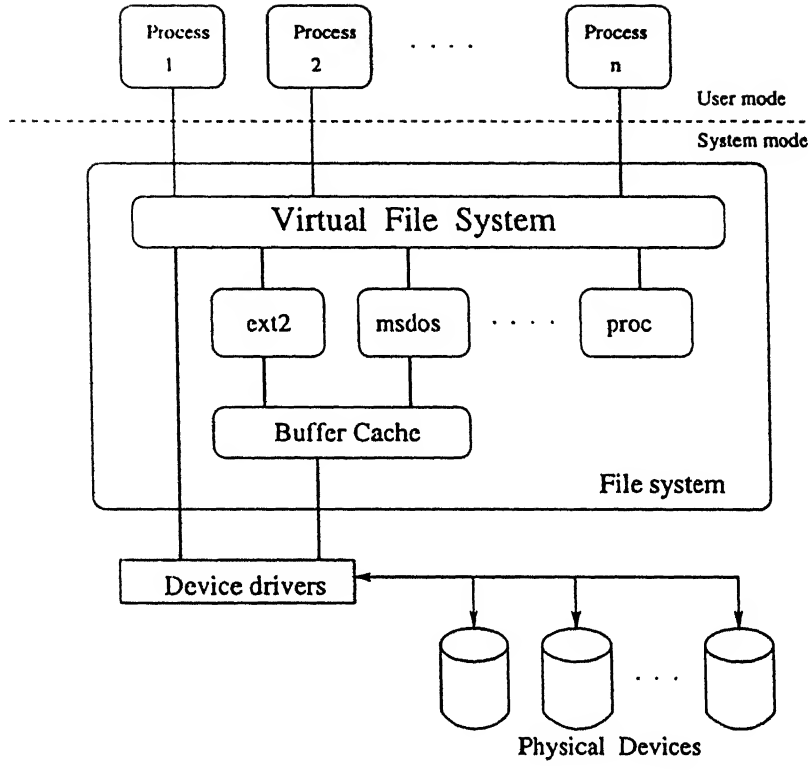



Figure 1: The layers in the file system

Like most modern variants of Unix, Linux also supports multiple file system types. This is achieved by having a unified interface between each of these different file systems and the Linux kernel. This interface is known as the virtual file system (VFS) [4]. The VFS interface was first introduced in SunOS in order to support NFS file system [6]. Figure 1 shows the various layers in the file system part of the Linux kernel code. The VFS layer provides a uniform view of file system hierarchy to the user. All file system related system calls are handled by the VFS layer. Upon receiving any such call, the VFS layer performs the file system independent portion of the call and then redirects it to the suitable file system module to perform the remaining portion. Apart from this, the VFS layer also manages internal structures such as super blocks, inodes and file structures. A more detailed discussion of these structures is given in Chapter 4. The buffer cache is a temporary storage in the kernel data area to cache recently used data, which improves system performance.

All the data transfer between the user processes and the file system takes place through the buffer cache. The device drivers are used to transfer the data from the physical devices to the buffer cache.

## 2.2 File Transfer Protocol (FTP)

In this section, we briefly describe the FTP protocol. For more details, refer to RFC-959 [5]. The File Transfer Protocol (FTP) provides an efficient and reliable way to transfer files from one computer to another. It was designed to be used either by a user at a terminal or by a program to control data transfer automatically.

### 2.2.1 The FTP Model

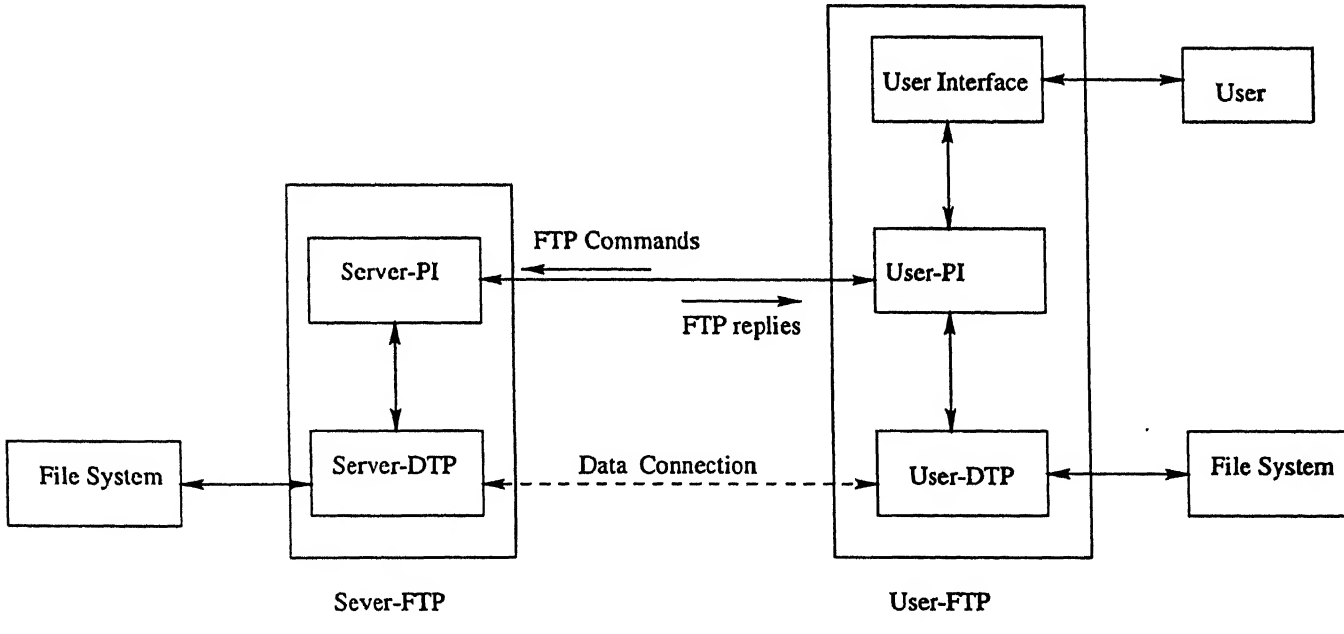



Figure 2: The FTP Model

The abstract model of the FTP service can be represented as shown in the Figure 2. The server-FTP is a process or a set of processes which perform the function

of the file transfer in co-operation with a user-FTP process. The functionality of the server-FTP can be divided into server-PI and server-DTP. Likewise, the user-FTP process can also be divided into user interface, user-PI and user-DTP. Upon invocation by the user, the user-PI process tries to contact the server-PI. The server-PI which is *listening* on a well known TCP port $L$ establishes the connection with the user-PI and this connection is called the *control connection*. Once the connection is established, all requests from the user are converted into FTP commands by the user interface and forwarded to the user-PI which sends them to the server-PI through the control connection. An FTP command is a string of characters whose meaning is understood by the server-PI and the user-PI and each command is associated with an action. These actions are performed by the sever-PI with the help of the server-DTP. If a request is for data transfer then the server-PI directs the server-DTP to establish the connection with the user-DTP which is *listening* on a port $D$ to transfer the data. This connection, called the *data connection*, exists only at the time of data transfer. The port number $D$ is either the default port number $L-1$, or informed by the user-PI before the data connection is established. In order to achieve synchronization between the requests and the actions for these requests, the server-PI sends one or more replies to the user-PI. The FTP-reply is an acknowledgment (either positive or negative) for the requested action. The format of the FTP-reply consists of reply code which is useful for the automated programs and the text followed by the code for humans.

### 2.2.2 The FTP commands

The FTP commands are divided into, the access control commands, the transfer parameter commands and the FTP service commands. The access control commands allow the server-PI to identify the user. This is necessary so that the server can determine whether the user is authorized to access the requested file. The *user*, *pass* and *quit* commands are examples of the access control commands. The *user* command is used to identify the user name and the *pass* command is used to specify the password of the user. The server-PI determines the accessibility of files by

the user by using the information provided by the *user* and the *pass* commands. The *quit* command is used to say that all requests are over. The transfer parameter commands are used to specify parameters to transfer data through the data connection. The commands in this class are *mode, type, stru* and *port.* The *mode* command specifies how the bits of data are to be transferred. The *type* and the *stru* commands are used to define the way in which data is to be represented. The *port* command is used to specify a port number other than the default one for the data connection which is used by the server-DTP to establish the data connection. The FTP service commands define the file transfer or the file system function requested by the user. The commands in this class are *retr, stor, dele, mkd, rmd, list* and *nlst.* The *retr* command is used to transfer a file from the server-FTP to the user-FTP. The *stor* command is used to transfer a file from the user-FTP to the server-FTP. The *dele* command is used to delete an existing file on the server file system. The *mkd* and *rmd* are used to create a new directory and to remove an existing directory on the server file system respectively. The *list* and the *nlst* commands are used to list a directory which belongs to the server file system. The format of the output produced by the *list* command is suitable for a human and may vary from one host to another. But for the *nlst* command the format is fixed and consists only of the file names which are separated by new line character. This command is useful for programs that automate data transfer.

## 2.3 Anonymous FTP

As mention in the earlier chapter, archive sites are hosts in the Internet which act as repositories of information. Anonymous FTP [3] is the means by which archive sites allow general access to their archives of information. Archive sites create a new account *anonymous* which has a limited access to the archive site. Archive sites usually restrict the anonymous users to search only a sub-tree of its file system hierarchy. Most archive sites do not allow the anonymous user to upload (write) files into the archive sites. A sample ftp session of a user to an archive site, godavari, using the standard Unix program *ftp* is shown in Figure 3.

```
$ftp godavari
Connected to godavari.cse.iitk.ernet.in.
220 godavari FTP server (Version wu-2.4(1) Tue Aug 8 15:50:43 CDT
1995) ready.

Name (godavari:tatik): anonymous
331 Guest login ok, send your complete e-mail address as password.
Password:
230-Welcome, archive user!  This is an experimental FTP server.  If
230-have any unusual problems, please report them via e-mail to
230-root@godavari. If you do have problems, please try using a
230-dash (-) as the first character of your password -- this will
230-turn off the continuation messages that may be confusing your
230-ftp client.
230 Guest login ok, access restrictions apply.

ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for file list.
pub  usr  lib  welcome.msg
226 Transfer complete.

ftp> get welcome.msg
200 PORT command successful.
150 Opening BINARY mode data connection for welcome.msg (312 bytes).
226 Transfer complete.
312 bytes received in 0.032 seconds (9.5 Kbytes/s)

ftp> put somefile
200 PORT command successful.
553 somefile: Permission denied. (Upload)

ftp> quit
221 Goodbye.
```

Figure 3: A sample anonymous FTP session

# Chapter 3

# File System Design

The background concepts reviewed in the previous chapter are necessary for a better understanding of the design presented in this chapter. We start this chapter with the design goals and the approach followed to achieve these goals. This is followed by a discussion of some design issues. Next, the overall design that emerges by resolving the various design issues is finally presented.

## 3.1 Design Goals

Our aim is to provide a transparent Unix file interface for files present on archival sites. By this we mean that the user should be able to access files on the archival site as if they are on the local disk using familiar utility programs such as *ls, cp, cd* etc., without having to modify or even recompile these programs.

Another goal that we had set for ourselves was to achieve such a transparent access with reasonable system performance by reducing the usage of network to the maximum extent possible. This is because the network data access rate is lower than the disk data access rate. Hence, by caching the most recently used data, a significant reduction in network usage can be obtained which ultimately improves the overall system performance.

## 3.2 Our Approach

As discussed in the previous chapter, Linux supports multiple simultaneous file systems. The file system related code functionality of the kernel is divided into the file system independent part and file system dependent part. The file system independent part, called the virtual file system (VFS) layer interacts with the other part, through a well defined interface. This interface consists of a few well defined functions whose functionality depends upon the type of the file system. By providing a new meaning to these functions, a new file system type can be added to Linux. Using this facility, transparent access to files residing on remote archive sites can be implemented by a new file system type that implements the client side of the file transfer protocol. The code for this file system would translate requests to read files into FTP commands and communicate with FTP servers to actually get the required data. This is the approach that we have taken to allow transparent access to files residing on remote archival site. In the remainder of this thesis, we call this file system the FTP file system.

## 3.3 Mounting an FTP file system

In any Unix-like operating system, a new file system can be grafted on an empty directory of the current file system by calling the *mount* system call. This process is called mounting and the empty directory on which the new file system is added is called the mount point. Mounting an FTP file system will enables the users to access the files on remote archival sites. Mounting each archival site separately is not a very attractive option. This is because the user would not able to access an archival site unless it is already mounted since in Unix like operating systems ordinary users are not permitted to mount a new file system. Predicting user requirements in this regard is impossible and mounting all the archive sites in the world is clearly infeasible. A possible solution to this problem is to mount the FTP file system just once, for all archive sites. This may be done at the system startup time. This is the approach that we have taken.

With this approach, the problem is to specify the archival site on which a file resides. Note that this would not be a problem if a mount were done for each site as in the earlier option since the mount point would implicitly specify the archival site. We resolve this problem in the following manner. Suppose that an FTP file system is mounted on the directory /ftp, resulting in the directory structure shown in Figure 4. Just below the root of the FTP file system are directories each of which represents an archival site. These directories will in turn contain all the files that are present at that archival site. Thus to access file /pub/src/myfile on site cd1, one would use the path /ftp/cd1/pub/src/myfile.

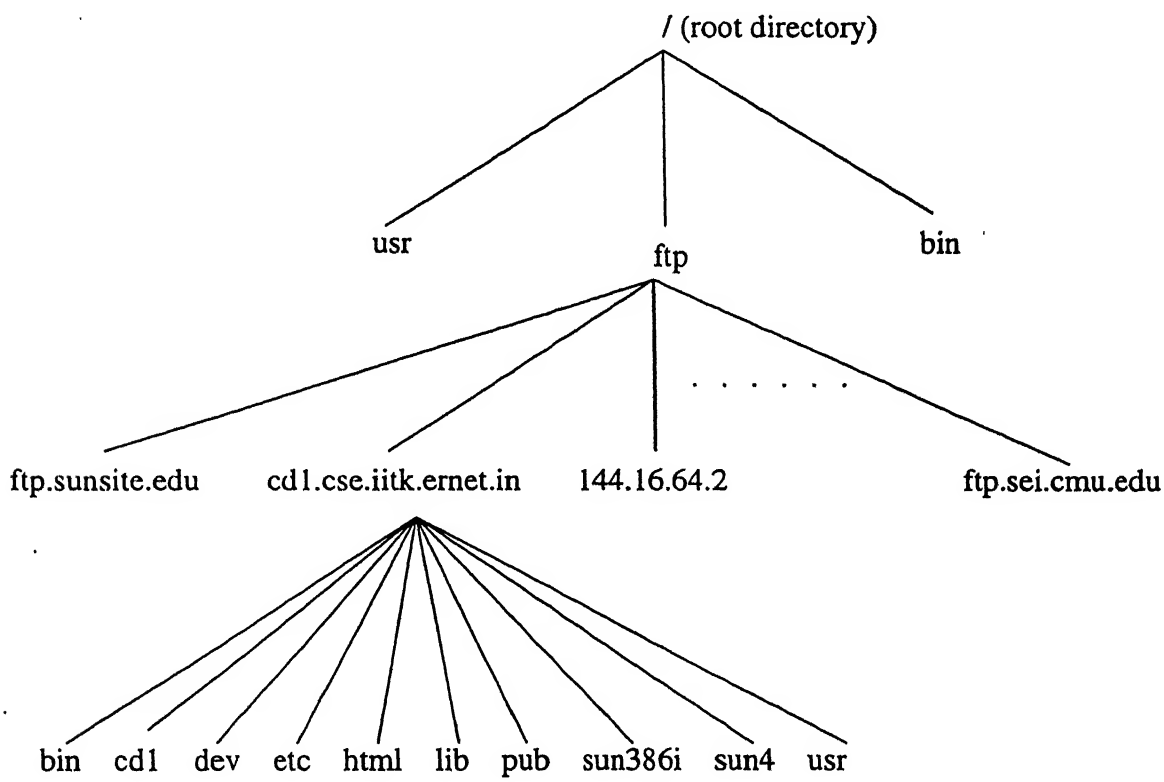



Figure 4: Logical view of the FTP file system hierarchy

Once the FTP file system is mounted, a user can access all the files of any archival site as if they were on the local disk. A user may then browse, using familiar Unix commands, the information on that archive site. Figure 5 shows a possible command

session wherein a user transparently accesses files on the archival site, cd1.

```
$cd /ftp/cd1

$ls -x
bin       cd1       dev       etc       html      lib       pub
sun386i   sun4      usr

$cp etc/group /local/group

$cat etc/group
wheel:*:0:
daemon:*:1:
post:*:11:
kmem:*:2:
bin:*:3:
tty:*:4:
operator:*:5:
news:*:6:
audit:*:9:
ftp:*:98:
ftpadmin:*:68:
```

Figure 5: A sample interaction with system

## 3.4 Problems with Non-Anonymous FTP

Anonymous FTP is the means by which information on an archive site is accessed through the special user account *anonymous*. By non-anonymous FTP we refer to accessing of files through ordinary user accounts. There are several problems in extending the FTP file system to allow non-anonymous access to FTP sites.

Accessing files using non-anonymous FTP requires the user login name and password. Thus, we need to design a secure way of providing the password of the user to the FTP file system. The only way the password can be specified is by either incorporating it in the path for the file or by storing it in some special file in the

user's home directory. Since the password has to be specified in un-encrypted form, both these methods would be insecure.

In contrast, accessing data on an archive site using the user account *anonymous* requires no password. Hence the FTP file system was designed for accessing data on archive sites using only anonymous FTP.

## 3.5 Caching

However the file transfer protocol only supports transfer of entire files. Transferring the entire file for each data block requested by the user would clearly be extremely inefficient. The only reasonable solution is to retrieve the entire file when it is first accessed and cache it for satisfying future requests. The cached copy can also be retained even after the file is closed by the user in anticipation of somebody accessing the same file again in the future. This would help in reducing the overall network traffic. It is relatively safe to cache data from an archive site since this data is not likely to change very frequently.

The cache can be maintained either in the main memory or in the hard disk. A main memory cache would be fast but can only be limited in size. Since we need to cache entire files, some of which may be very large, we choose to use the disk for maintaining the cache. Another advantage of a disk cache is that the contents of such a cache are not lost when the system crashes or is rebooted. We next discuss the design of the cache.

### 3.5.1 Cache Design

The design of our cache is very simple. We use a directory of some regular file system as the cache. This directory can be specified at the time of mounting the FTP file system. The cache directory has sub-directories, each representing an archival site. The directory name is the same as the canonical name of the archival site and the directory structure within this directories will mimic that of the root directory of the

archival site. Thus the name of the cache copy of any file in the FTP file system can be easily obtained. For example, if the cache directory is /cache and the canonical name of the server cd1 is csealpha1.cse.iitk.ernet.in, then the cached copy of the file /ftp/cd1/pub/src/x is in the local file /cache/csealpha1.cse.iitk.ernet.in/pub/src/x, if it exists.

### 3.5.2 Cache coherence

When a file in the FTP file system is opened the entire file is transferred from the archive site if it does not exist in the cache. After closing the file, the FTP file system does not remove the corresponding file in the cache with the anticipation that there will be future requests for this file which may be satisfied from the cache itself. This avoids the network transfer of the file. But this creates a situation where the FTP file system would supply outdated data to its clients if the data at the archive site changes. This state is known as cache incoherence. A partial solution to avoid this state is to remove the files in the cache after some time, which forces the FTP file system to acquire the data again from the archive site. The cache daemon is a continually running user level process which removes the files in the cache according to policies specified by the system administrator. A detailed discussion of the cache daemon is presented in Chapter 5.

## 3.6 Differences from Unix file semantics

There are some places where the semantics of our FTP file system differ from the usual Unix file semantics. The most glaring difference is that the *ls* command on the mount point will give an error because logically this directory contains all possible archive sites. Since we cannot list all archive sites in the world, we disallow the open system call for the root directory of the FTP file system.

Another problem is that FTP does not provide much information regarding a file except for its content, size and type. Hence, we assume reasonable values for the remaining file attributes such as file owner, group, permissions, date of last access

and modification etc. Thus it may happen that *ls* shows a file to be readable but it actually is not.

## 3.7 Overall Design

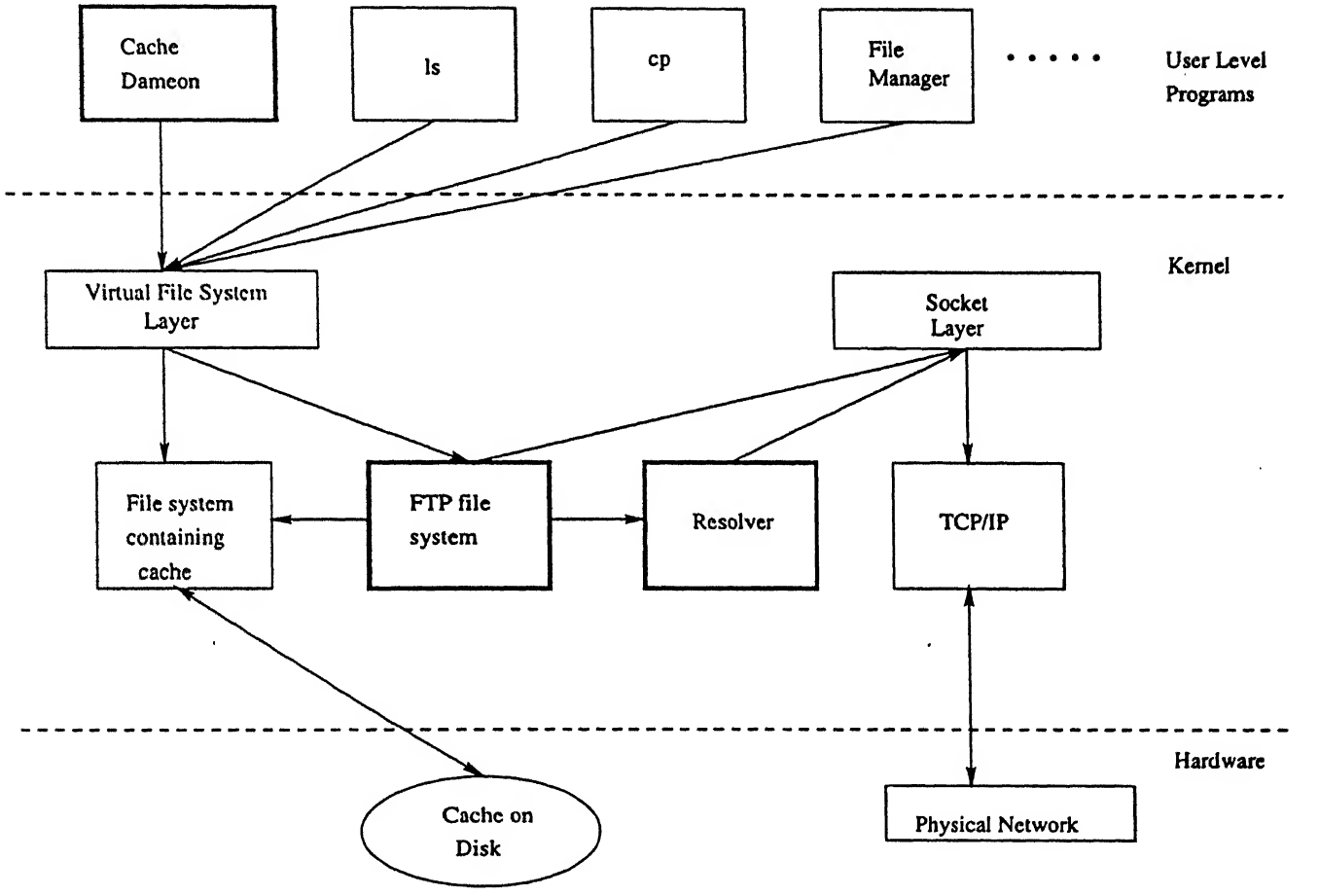



Figure 6: Design of the file system

The overall design that emerges from the above discussion is shown in Figure 6. Only the modules enclosed in thick lines need to be implemented and the rest are already available as part of the kernel. Of these, the FTP file system and cache daemon have been already discussed and their implementation is described in detail in Chapter 4 and 5 respectively. The remaining module, resolver module, converts host names into its 4-byte IP addresses by sending a request to name servers, which make the suitable conversion. This is required because IP addresses are needed to access archive sites but we expect users to provide only the host name. Although the existing user library function *gethostbyname* does this, we cannot directly make use of its functionality as it is in user address space. Hence, we are forced to re-implement it in the kernel.

# Chapter 4

# Implementation of the FTP file system

In order to provide the Unix file interface to archive sites a new file system, FTP file system, was designed. The design of this file system was discussed in the previous chapter. The FTP file system was implemented as a new type of file system for the Linux operating system. In this chapter the implementation of this file system is discussed. Before we present the implementation details, the internal data structures of the VFS layer are discussed for a better understanding of the implementation of the FTP file system.

## 4.1 Internal Structures of VFS Layer

In order to have a uniform interface between the VFS layer and different file systems, the VFS layer assumes that each file system has the same structures to manipulate files. These structures are: super block and inodes. The super block contains information about the entire file system. Inodes contain information about individual files. Both these structures have three main components. They are the file system type independent fields, file system type dependent fields and a vector of pointers to functions. These functions form the interface between the VFS layer and file system

dependent modules.

The super block has fields like block size, access rights, time of last change etc., in the file system independent part. The other important file system independent fields in this structure are *s_covered* and *s_mounted*. The field *s_covered* is a pointer to the inode on which this new file system is being mounted and the field *s_mounted* is a pointer to the root inode in the mounted file system. The file system dependent part usually contains information such as the number of blocks in the file system, the number of free blocks, the number of inodes and the free inodes.

All the files are represented as inodes in the kernel and these inodes are stored in an inode cache which is a part of the kernel data area. This cache is managed by the VFS layer functions *iget* and *iput*. The file system independent part of an inode contains fields like owner, group etc., of the file, number of links to this file, various access times, permissions and other information required to manage these inodes. The file system dependent part may have the disk block numbers where the data of the file resides.

Any file system is required to provide functions for super block operations, inode operations and *file* structure operations. The super block operations manipulate and give information about a file system as a whole. These operations are known to the VFS layer at the time of mounting the file system. The super block operations are applied to read or write an individual inode of any file and to obtain the information about the entire file system. At the time of mounting the file system, the VFS layer is also informed about the root inode of the mounted file system. By using the *lookup* operation of the root inode, the VFS layer can determine the inodes of the files in the next level in that file system and so on. Thus, it is able to get all the inodes of a file system. Once the inode of a file is obtained, all further operations can be performed by using the operations in this inode. These inode operations are applicable only to this inode and provide the functionality to manipulate the file system hierarchy. These operations are stored in a structure and each inode has a pointer to this structure. This structure also has a pointer to the *file* structure operations which are to used provide a general interface to access a file. These *file* structure operations may be changed for a file at the time of opening it.

## 4.2 Mounting the FTP File system

```
function:   read_super_block.
parameters: pointer to the super block, mount options.
begin
     if FTP file system is already mounted then return error.
     extract the cache directory name from the mount options.
     if the cache directory does not exist then
        create the cache directory.
     get the root inode of the FTP file system.
     set the s_covered and s_mounted pointers in the super block.
     Allocate memory and copy the cache directory name.
     get the global information required for resolving host names.
     return filled super block.
end
```

Figure 7: The read_super_block function

For accessing a file in any file system, the file system on which the file resides must be first mounted. During mounting a new file system, the VFS layer calls the function *read_super_block* of that file system to read the super block information.

For the FTP file system, the file system specific part of the super block contains the name of the cache directory and a pointer to the super block of the file system containing the cache directory. These are necessary to access the files in the cache. The following subsection describes the *read_super_block* function of the FTP file system.

### 4.2.1 The Read_super_block Function

The main functionality of this function is to fill up the super block structure and to get the root inode of the mounted file system. In our case, apart from these tasks this function will make sure that the cache is ready for use. The main algorithm is shown in Figure 7.

The function *read_super_block* takes a pointer to the super block and mount

options as the arguments and it first checks for the presence of another FTP file system which may have been be mounted already. If the FTP file system is already mounted then it returns an error. Thus mounting the FTP file system twice is not allowed. The name of the cache directory is extracted from the mount options to check for its existence in the local file system. In case it does not exist, a new directory is created in the local file system. The root inode of the FTP file system is obtained by calling the function *iget* with the inode number as an argument. Prior to this, the inode number of the cache directory is determined. This is because the FTP file system assumes that its root inode number is same as that of the cache directory inode number. The fields *s_mounted* and *s_covered* are filled with the suitable inode pointers to traverse the file system hierarchy smoothly.

As described in the previous chapter, the FTP file system uses the function *gethostbyname* to convert the machine name into an IP address. This function uses the configuration file */etc/resolv.conf* to initializes its structure which contain the information about the name-servers. This structure is called *__res_state*. This operation has to be done once for the entire file system and before accessing any file on the remote host. Therefore *read_super_block* function of the FTP file system calls the function *res_init* to initialize this structure. Finally, the filled super block is returned to the VFS layer.

## 4.3 The File system Specific Information in the FTP File system Inodes

In order to understand the super block operations it is necessary to know the file system specific information of an inode because some of the operations work on this structure. Hence, in this section we discuss more about the file system dependent fields in an inode of the FTP file system.

Once a file system is mounted, all the files in that file system can be accessed. Since all files are represented as inodes in the kernel, all path names are converted into inodes. As mentioned earlier, an inode contains information about a file and

```
struct ftp_inode {
        char                *in_cache_path;
        char                *not_in_cache_path;
        int                 in_cache;
        struct  inode       *cache_i;
        struct  inode       *opened_inode;
        struct  file        *opened_file;
        struct  cache_host  *host;
}
```

Figure 8: FTP file system specific part of the inode

this information is divided into the file system dependent and independent parts. In case of the FTP file system this file system dependent information contains fields shown in Figure 8.

The fields *in_cache_path* and *not_in_cache_path* together contain the total path name of a file on the remote machine which is represented by this inode. The path name is divided into two parts depending upon the presence of its components in the cache. The first few components in the path name, which are in the cache, are stored in the field *in_cache_path* and the remaining path name is stored in the field *not_in_cache_path* for the sake of convenience. The path names are necessary because FTP requires them to transfer files from a remote machine.

As said previously the first component of the path after the mount point is required to be a host name and each host is represented as a sub-directory in the cache directory. Each of these sub-directories will have the same hierarchy as on the remote host. In effect each component after the mount point corresponds to a file or a directory in the cache. The field *cache_i* is a pointer to the inode of the corresponding file in the cache. For the root inode of FTP file system, this field points to the inode of the cache directory.

The field *host* is a pointer to the structure containing the information about the host on which the file actually resides. This information includes the host IP addresses and the other names of the host. The IP addresses of a host are

necessary to access it. The information in the *host* structure is filled by the function *gethostbyname*.

The field *in_cache* is a flag used to indicate whether this file is in the cache or not. The remaining fields *opened_inode* and *opened_file* are explained later in this chapter.

## 4.4 Super block operations

As said previously the super block structure contains a vector of functions. These functions are *read_inode, write_inode, put_inode, notify_change, put_super, write_super* and *statfs*. The implementation of the functions *read_inode, put_inode* and *put_super* is discussed in the following subsections.

The functions *write_super* and *write_inode* are used to save the information of the super block and inode on to the disk. The function *stat_fs* gives the information about the entire file system like number of free inodes and free blocks. For the FTP file system these functions are not required because this is only a logical file system which does not exist on the disk. Hence, these functions were not implemented.

Usually inode information is changed in the system calls and these are informed to the file system by calling the function *notify_change*. In the FTP file system this call simply gets redirected to the inode of the corresponding cache file because the inode information of a FTP file system file is maintained in the inode of the corresponding cache file.

### 4.4.1 The read_inode Function

As said previously, the function *iget* is used to acquire any inode. Its arguments are the required inode number and the super block of the file system to which this file belongs. This function first checks for the inode in the inode cache. If the inode is present then it returns it after incrementing the reference count. Otherwise, it allocates memory space from the free list of inodes. To fill in the file system

dependent information, it calls the file system specific *read_inode* function. The argument for this function is an inode with the inode number and information about the super block filled in. This function fills up the other fields before returning.

In our case, this function again uses the function *iget* to read information about the inode of the corresponding cache file. We always assign the same number to an inode of the FTP file system as the inode number of the corresponding cache file. Thus the inode number of the cache file is easily known. Here it uses the cache file system's super block, which is stored in the file system specific information of the FTP file system's super block. By using the information in the cache inode it fills some of the fields in the FTP file system inode such as *owner, group, permissions, type* and *size*. It also allocates memory required for the field *in_cache_path* to store the path name and supplies pointers to functions that implement the inode operations.

### 4.4.2 The put_inode Function

This function is called by *iput* if the inode is no longer required. Its main task is to delete the file physically and release its blocks if the number of links to this inode is zero. Since FTP file system does not have any associated physical devices, it does nothing in this regard. But it makes use of this opportunity to release the memory allocated for the field *in_cache_path*.

### 4.4.3 The put_super Function

This function is called when the file system is unmounted. It releases any resources that are associated with this file system. In our case, it releases the memory allocated for the cache directory name in the *super block* structure. This also releases the memory allocated for the structure *__res_state* because it is no longer required.

## 4.5 Inode operations

Each inode contain pointers to functions that implement operations on the inode. These functions provide functionality which is useful to manipulate the file system hierarchy. Typical functions are *create, lookup, link, unlink, symlink, mkdir, rmdir, mknode, readlink, follow_link, bmap, truncate, permission* and *rename.* The FTP file system is not allowed to add a new file or delete a file to the existing file system because it implemented as a read-only file system. Hence, all the functions which involve creating a new file or deleting an existing file are not necessary which is the reason why the functions *create, link, unlink, symlink, mkdir, rmdir, mknode, rename* and *truncate* were not implemented. Since the FTP file system does not support symbolic links, the functions *read_link* and *follow_links* are not necessary and were also not implemented.

The function *permission* is used to check for the access rights of the calling process for an inode. As FTP file system allows any user to read or execute any file this function simply implements this policy.

### 4.5.1 The lookup Function

The initial access to a file is by its path name, as in the *open, chdir,* or *link* system calls. Because the kernel works internally with inodes rather than with path names, it converts the path names into inodes to access files. The function *namei* parses the path name one component at a time, converting each component into an inode by calling the *lookup* function, and eventually returns the inode of the path name. Since this function depends upon the file system type, the VFS layer redirects the request to the function *lookup* of the suitable file system.

The *lookup* function takes a file name and directory inode as its arguments and returns a pointer to an inode in the inode cache. The inodes supplied by *lookup* should be such that all inodes of a single file system have different inode numbers. To satisfy this requirement each file system maintains a mapping between inode numbers and path names. In a normal file system this mapping is stored in

```
Function:     lookup
Parameters:   file name, parent directory inode, last component.
begin
    if the parent directory is the mount point of this file system
    begin
      determine the canonical name of the host represented by file.
      if the directory with this canonical name is not present in
                                    the cache  then
          create a directory in the cache with this canonical name.
      return the new inode by filling appropriate information.
    end
    else if the parent directory is not present in the cache
        add file name to not_in_cache_path of the parent directory.
    else if the file name is not present in the cache
    begin
        lock the parent directory inode.
        add file name to not_in_cache_path of the parent directory.
    end
    else
        return the new inode by filling appropriate information.
    if this file name is not the last component of the path
        return the same parent directory inode.
    if this path name is not present on the remote host then
        return error.
    for each component not_in_cache_path
        if this component is not the last then
            create the directory.
        else
            create file or directory as appropriate.
    unlock the parent directory.
    namei(parent directory, not_in_cache_path, resultant inode).
    return the resultant inode.
end
```

Figure 9: The Lookup function

directories. In our case the remote host does not supply these inodes numbers and even if it did supply them, they would not necessarily satisfy the above requirements. This is because it is possible that two different files which exist on different hosts



CENTRAL LIBRARY

may have the same inode number. A possible solution for this problem is to use the local file system assigned inode number of the cache file as the inode number of the FTP file system file also. This requires the *lookup* function to create the file in the cache before assigning a inode number to the FTP file system file, if it does not exist. The overall functioning of this function is shown in Figure 9.

This function treats the first component after the root of this file system as a special case. In this case, the file name, which actually indicates a machine name, is converted into a canonical name to check for the presence of the corresponding directory in the cache. This is necessary because all the hosts are represented as directories in the cache with their canonical names. To affect this conversion the function *gethostbyname* is called. As a side effect the IP address of the host is returned which can be used to contact the remote host. Hence, this information is stored in the field *host* of the inode and it gets passed to all the inodes of the files that are below this directory in the file hierarchy.

If the corresponding directory is not present in the cache then a new one is created. A new inode is returned after filling in the fields appropriately. The FTP file system requires the inode of the corresponding cache file because some of the subsequent requests to this inode will be redirected to the cache inode. Hence, it is necessary that the corresponding cache inode is in the inode cache till the FTP file system inode is removed from the inode cache. To do this, the reference count of the cache inode is increased by one whenever the reference count of the FTP inode goes from zero to one and it is decreased by one whenever the reference count of the FTP file system inode drops to zero from one. This addition is done in the function *read_inode* and the corresponding reduction by one is done in the function *release_inode*.

If the component is not the first one after the root of the FTP file system, a check is made for the presence of the file in the cache. If it exists, the same inode number as that of the corresponding cache file is assigned to this file and a new inode is returned after filling in the required information. Otherwise, the remote machine has to be contacted to check for the presence of the file on that host. If it does exist, a corresponding file should be created in the cache and the inode should be returned.

However doing this for every component would be very inefficient. Therefore, if this component is not the last one in the path, this function simply stores the component name in the field *not_in_cache_path* in the parent directory inode and sets its *in_cache* field to one to indicate that the remaining components in this path are not in the cache and that their names should simply be appended to *not_in_cache_path*. It then returns the parent inode itself after locking it. Upon receiving this inode the VFS layer again sends a requests to this function by giving the next component and the same inode. But this time by looking at the field *in_cache*, the function simply appends this component name to the field *not_in_cache_path*. This process is repeated till the last component is reached.

Upon reaching the last component, the remote host is contacted to check for the presence of the pathname on it by calling the function *whatisit*. If this function returns an error then this means that the file does not exist. So, the *lookup* function releases the inode lock and resets the field *in_cache* to the correct value. After this an error is returned to the VFS layer. In case the path name exists, a directory is created for each component except for the last one of the path name stored in the field *not_in_cache_path*. For the last component a file is created depending upon its type i.e, either a *regular file* or a *directory*. The function is now ready to return the inode of the path name. So the inode of the path name is returned but before doing so it releases the lock on the directory inode and resets the field *in_cache* to its correct value.

### *The whatisit Function*

The function *whatisit* establishes the command connection with the remote host and logs on as an *anonymous* user. To check the existence of the path name it uses the FTP command *nlist* which expects the path name as an argument. If the remote host returns an error code this means that the file does not exist. Hence, *whatisit* returns an error to *lookup*. If the same path name is returned by the remote host then this path name represents a regular file. In this case, *whatisit* returns a flag indicating that the given path name is a *regular file*. Otherwise it returns a flag to

say that the path name represents a directory.

## 4.6 File operations

The *file structure* also has a vector of functions, which are held in the field *f_op*. These functions are used to provide the general interface to access a file. The typical functions are *open, read, write, readdir, lseek, fsync, select, ioctl* and *mmap*. The functions *write* and *fsync* are related to writing a file. Since the FTP file system is implemented as read-only it does not require these functions and hence they were not implemented.

The VFS layer provides the default operations for some of the functions like *lseek* and *mmap* which are used if a file system does not implement these functions. However, if a file system wants functionality other than the default one, it has to implement the function suitably. The default functionality of *lseek* is to move the file pointer within the file. The file pointer points to a data byte in the file from where all the subsequent read or write request will start transferring data between the user buffer and the kernel buffer. The default functionality of *mmap* function is to map a part of the file to the user address space of the process. The FTP file system also requires the same functionality. Hence, we did not implement these functions.

The function *select* checks whether data can be read from a file or written to it without any delay. This is useful only for device files and sockets. In case of regular files the VFS layer provide the default operation which is to simply return a value indicating that the data is ready. We did not implement this function because FTP file system does not support sockets and device files.

The other functions *open, read, readdir* and *release* are implemented and we discuss them separately in the following paragraphs.

### 4.6.1 The open Function

This function gets called when a user tries to open a file. The main functionality of this function is to ensure that the access to data is without problems. In our case, the implementation of this function is shown in Figure 10.

```
function:   open
parameters: pointer to inode of to be opened file and file structure

begin
    if the given file is a directory then
    begin
       if the corresponding cache file ".dir" does not exists then
          create a ".dir" file in the corresponding directory.
       if the file ".dir" does not have any data then
          getfilelist(inode). /* This will get the data from the
                                                      remote host   */
   end
   else if the corresponding file does not contain data then
      getdata(inode).         /* This will get the data from the
                                                      remote host   */
   allocate a new file structure.
   initialize this structure and return.
end
```

Figure 10: The open function

As said previously, to simulate random access over sequential access the FTP file system ensures that the entire file is in the cache before the first request to access it is received. The *lookup* function ensures only the existence of the corresponding file in the cache but not about the data in that file. Hence, this function first makes sure that the data is also present in the cache. The test for data existence in the cache file is done using its owner field. If the owner of the cache file is *ftpdnr* (*FTP Data Not Ready*) the data is not present in the cache. Otherwise, it is present in the cache file and its owner is *ftpdr* (*FTP Data Ready*). When a file is first created in the *lookup* function, it is created with the owner as *ftpdnr* and its ownership is changed to *ftpdr* after the data of this file gets transferred from the remote host in

the *open* function.

If the data in the corresponding cache file is not present then the function *getdata* is called to transfer the contents of the file from the remote host. In case of a directory, it checks for the existence of the file *.dir* in the corresponding cache directory. This file by convention contains the listing of this directory. This is required since all the files in this directory may not be in the cache. If the *.dir* file does not exist then it is created and the function *getfilelist* is called to fill the file with the listing of the directory from the remote host on which this directory exists. The file names in the file *.dir* are separated by a new line.

Each opened file has a *file* structure in the file table which has a pointer to the inode of the file and a pointer to this *file* structure is stored in the file descriptor table of the process. The file descriptor, which is an index of a pointer to the *file* structure in the file descriptor table, is returned to the user process as the return value of the open system call. The user process is required to supply this file descriptor as an argument in the subsequent requests to access this file. Using this file descriptor the VFS layer is able to access the *file* structure and the inode either to redirect requests to the suitable file system or to use the data in these structures. Since the FTP file system satisfies all read requests from the corresponding cache file, it has to keep track of the *file* structure and inode for the cache copy of an open file. Hence, after ensuring that the required data is present in the cache it allocates a *file* structure in the file table and stores a pointer to this *file* structure in the field *opened_file* of the FTP file inode. A pointer to the inode of the corresponding cache file is also stored in the field *opened_inode*. This means that the inode represented by *opened_inode* will be either a *.dir* file if the opened file is a directory or a corresponding file in the cache if the opened file is a regular file. After allocating the *file* structure it initializes the file pointer to the first byte of the file and other fields to suitable initial values. Now it is all set to access the data without any problem.

- *The getdata Function*

This function gets the contents of a file from the remote host. The inode of the file is a parameter to this function. It establishes a connection to the FTP server on the remote host. It then sends the FTP command *get* with the path name of the file saved in the *in_cache_path* field of the inode as an argument to the remote host. Upon receiving this command, the remote host establishes a data connection with this host and starts transferring the data, which is stored in the corresponding cache file. After completion of the data transfer the function returns.

- *The getfilelist Function*

The functionality of this function is also the same as that of *getdata* except for the FTP command which is used. The *nlist* command is used here instead of *get*. The argument to this command is the name of the directory which is stored in the field *in_cache_path* of the inode.

### 4.6.2 The readdir Function

This function returns the next directory entry in the *dirent* structure which is a parameter to this function. This structure contains fields like name of a file and an inode number. In our case, it reads the next directory entry from the corresponding *.dir* file and fills the name of file in the *dirent* structure. This function supplies an arbitrary number as the inode number because the file may not exist in the local cache.

### 4.6.3 The read Function

This function copies the requested number of bytes of data from the file to the user buffer. In our case, the request is redirected to the inode of the corresponding cache file which supplies the required number of data bytes.

### 4.6.4 The release Function

This function gets called when the file is closed, which was opened previously to access a file. The main functionality of this function is to release resources, which were allocated at the time of opening the file. In our case, it releases the *file* structure which was allocated to get the information about the corresponding cache file.

# Chapter 5

# The Cache daemon

As mentioned earlier, caching is used to improve the performance of our system and to simulate random access over sequential access. Thus the FTP file system gets the entire file from the remote site before satisfying the first read request for it and all the subsequent requests are satisfied from the cache itself. Once the data is present in the cache the FTP file system satisfies the requests from the cache. Hence, if the data at the FTP site changes, the FTP file system would still supply old data from the cache. This state of supplying outdated data from the cache is traditionally known as cache incoherence. Since the FTP servers do not inform about the data modifications to the client, a general solution to this problem is hard to achieve. A partial solution to avoid cache incoherence is to delete files from the cache after some time thus forcing the FTP file system to acquire the data again from the server. This removal of data can be done either by the FTP file system i.e., inside the kernel or by a user level process i.e., outside the kernel.

A policy i.e., after how much time and which files are to be removed, is highly dependent upon the archive sites because the frequency at which data changes for different archive sites varies widely. Hence, designing a single policy which captures the data changes of all archive sites is infeasible. Thus, a program which attempts to remove cache incoherence should be able to change its policy in accordance with the characteristics of various archive sites. Hence, such a program should be flexible

enough that its deletion policies can be easily configured. Implementing such a program in the kernel is clearly undesirable because changing the kernel code is a difficult task. Another solution is the design of a user level process which implements a policy in accordance with the user's requirements. It would be very easy to modify such a program to implement a new policy. In the following sections we describe the design and implementation of such a program that we implemented. This user level process is called as the *cache daemon.*

## 5.1 Design of the Cache Daemon

The cache daemon is a user level process which runs continuously. The system administrator can easily configure it to adopt policies of the following kind: "remove all files whose path names match regular expression $r$ and which were brought into the cache at least $y$ time ago and have not been accessed for time $z$". Hence, the cache daemon provides a way to specify a *class* of file names and two other options with regard to access time and creation time. These time options are used to specify the removal of all the files which were not accessed since the specified time and were brought into the cache before the specified time. If either of the time options is not specified the cache daemon assumes that all files satisfy that condition. At least one time option must be specified. Any number of such policies can be specified. A policy such as "remove files which were created one week back or which have not been accessed since two days" can be specified as two separate policies each of which specifies one type of time option. In later sections the exact syntax for the specification of policies is described.

Running this process with the owner *ftpdr* precludes the possibility of accidental removal of other files. The cache daemon is able to remove files in the cache because all the files in it are created with the owner either as *ftpdr* or as *ftpdnr* and the permissions for these files are read, write and execute for the owner of the file and group. The users *ftpdr* and *ftpdnr* are in the same group: *ftp*.

## 5.2 Implementation

```
Function:     Cache Daemon.
Parameters:   Command line arguments and configuration file.
begin
    make this process as a daemon.
    parse the command-line arguments.
    Get the policies from the configuration file.
    while (True)
    begin
        Traverse(Start directory of the Cache);
        sleep for a while.
    end
end.

Function:         Traverse.
Parameters:       Directory name.
begin
    for each file in the input directory
    begin
        if the file name is matched in any policy and time
        conditions are also satisfied then
        begin
            if it is a regular file then
                remove this file.
            else if is a directory
                remove all files in this directory and itself
        end
        else if this is a directory then
            Traverse(directory).
    end
end
```

Figure 11: Cache daemon and Traverse Functions

The algorithm of the cache daemon is shown in the Figure 11. Like other standard Unix daemons, it does routine jobs such as closing all the opened files, resetting the file creation mask to zero and disassociating itself from the process group. It then parses the command line options and stores the specifications given by the

users. It reads policies from the standard configuration file */etc/ftp_fs.conf* or any other user specified file which may be specified through a command line option. The current working directory of the daemon is changed to the starting directory of the cache. During the traversal of the directory tree, at every node the cache daemon tries to match the file name with the regular expressions of all policies. If the file name matches and its type is regular then it is removed. If the matched file is a directory then the cache daemon recursively removes all files in that directory and the directory itself. The cache daemon omits the file *.dir* in matching the file names with the regular expressions. This is because the file *.dir* is transparent to users of the FTP file system. After traversal of the entire directory tree, the cache daemon sleeps for some time and this time interval is specified by the user as one of the command line arguments. This ensures that the cache daemon does not hog resources.

## 5.3 Specification of the Configuration File

A sample configuration file is shown in Figure 12. In this file each line specifies either a comment line, a policy line or a command line. All lines that start with the character '#' are considered as comments. Each policy line consists of one or more options where each option is of the form *option-name = expression.* The options may be specified in any order. All white spaces are ignored. The option-names in the policy line are *path, apath, atime* and *ctime.* The option-name *path* is to specify the file names with respect to the current directory and the option name *apath* is to specify the file names with respect to the starting directory of the cache. The *expression* for these option-names is a string which is enclosed between double quotes. This string is a shell type regular expression. The option-name *atime* is to specify a time period and if the difference between the current time and last access time of a file is less than this value then the file is selected. The option-name *ctime* is to specify a time period and if the difference between the current time and creation time of a file is less than this value then the file is selected. If the selected file satisfies other options in the policy then it is deleted. The *expression* for these

```
#The following line is to specify a policy that remove all files
# that were not accessed since one day and created two days ago.
path = "*" atime = 86400 ctime = 17280

#The following line is to specify  a policy that remove all files
# from the host cd1.cse.iitk.ernet.in and which were not accessed
# since a week.

atime = 604800 apath = "cd1.cse.iitk.ernet.in/*"

#The following line is to specify the name of the starting cache
# directory
cache = "/cache"

#The following line is to specify the time gap between two
# consecutive tree traversals.
sleeptime = 86400
```

Figure 12: A sample configuration file

time options is an integer specifying the value of the time period in seconds. Each command line consists of a single statement which has the same format as that of the above specified option statements. The option-names for command lines that are supported by the cache daemon are *cache* and *sleeptime*. The option-name *cache* specifies the starting directory of the cache and its *expression* is a path name enclosed between double quotes. The option-name *sleeptime* is the time gap between two consecutive traversals of the cache directory and its value i.e., an *expression*, is an integer to represent the value of time in seconds.

## ▮ *Command line arguments*

The user can also give the command-line options to the cache daemon process. The following are the options to this daemon:

-f to specify a configuration file different from the default one.

-c to specify the cache directory.

-s to specify the sleep time.

# Chapter 6

# Results and Conclusions

Our main aim was to provide an easy and better interface to archive sites. To measure the performance of the FTP file system, a few benchmark experiments were conducted. The same experiments were also conducted for standard software such as *rcp, ftp* and *NFS* to compare their performance with that of the FTP file system. In this chapter, we present the results of these experiments. Finally we make some concluding remarks.

## 6.1 Performance Experiments

Since the basic functionality of the FTP file system is to provide file transfer, we measured the time taken for transferring files of various sizes from an archive site to the local host. Both the machines were on the same 10Mbps Ethernet LAN.

We measured the time taken to copy files of various sizes from the archive site to the local disk using *rcp [7], NFS [6], the standard FTP client [1] of Linux* and *the FTP file system.* The results obtained are tabulated in the Table 1 and graphically shown in Figure 13. For the FTP file system, the two cases — when the remote file is in the cache and when it is not in the cache were considered separately and the times for both are shown in Table 1 and Figure 13.

It can be seen that if the file to be copied is in the cache, the FTP file system

| Size of file | NFS | RCP | Standard FTP client | FTP file system when the file was not in cache | FTP file system when the file was in cache |
|---|---|---|---|---|---|
| 4K | 0.178916 | 1.023283 | 0.761448 | 1.508677 | 0.162840 |
| 8K | 0.209748 | 1.266267 | 0.792632 | 1.560243 | 0.180969 |
| 16K | 0.249970 | 1.250892 | 0.805916 | 1.521680 | 0.187067 |
| 32K | 0.471126 | 1.221384 | 0.914988 | 1.497572 | 0.181172 |
| 64K | 0.663196 | 2.081319 | 0.821936 | 1.621062 | 0.211081 |
| 128K | 0.891423 | 1.710177 | 0.894391 | 2.158447 | 0.250539 |
| 256K | 1.674775 | 2.674244 | 2.318634 | 2.847186 | 0.404060 |
| 512K | 3.528790 | 5.566256 | 4.793826 | 4.219165 | 0.524091 |
| 1024K | 6.381463 | 10.521070 | 5.628077 | 9.782810 | 1.124221 |
| 2048K | 15.956506 | 20.375831 | 19.457530 | 26.126051 | 6.223572 |

Table 1: Comparison of file transfer times (all times shown are in units of seconds)

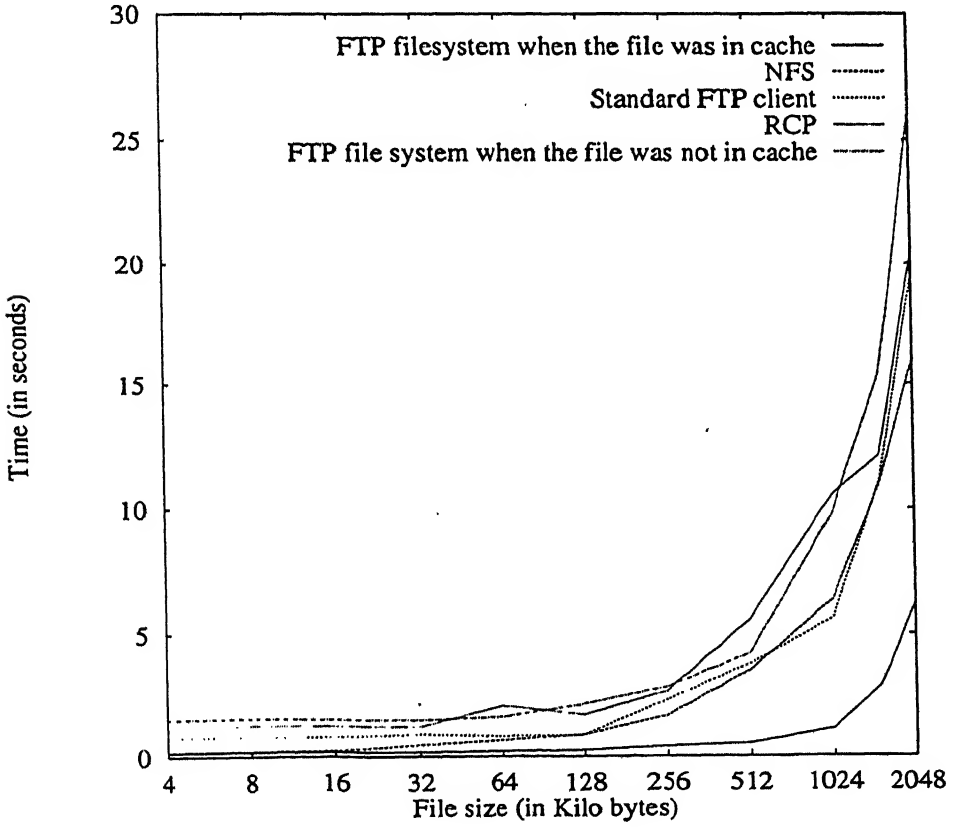



Figure 13: Comparison of File transfer times

outperforms all other software. This is clearly so since in this case, the file is simply fetched from the local disk. When the file is not in the cache the FTP file system performs worse than other software. This is because of the extra overheads of the FTP file system such as verifying the existence of the file before actually getting its contents.

## 6.2 Conclusions

In this report, we have described the design of a new file system which provides a transparent local file like interface to files on an archive site. It was implemented as a new file system on the Linux operating system. Our design incorporates caching of remote files on persistent storage in order to to improve its performance. We use a user level cache-daemon process to remove files from the cache in order to reduce the problems of cache incoherence. The policies for removing files from the cache can be specified by the system administrator in a very flexible manner.

In order to measure and compare the performance of the FTP file system with that of other file transfer programs, a few experiments were conducted. As expected the FTP file system performs poorly when the requested file is not in the cache and it performs better than other software when the required file is in the cache.

## 6.3 Future Work

In the current version, the FTP file system was implemented as a read-only file system because many sites do not allow the *anonymous* user to write onto the archive site. There are a few sites which do allow the *anonymous* user to add new files to the archive site without permitting changes to the existing ones. Adding new files to the archive site requires that the file system should support the write operations. By implementing the functions which are related to writing files, the FTP file system can be extended to add new files to archive sites. The current implementation is probably not an optimal one. Performance tuning can be undertaken in order

to improve its performance.

# Bibliography


[1] On-line Manual for ftp.

[2] BECK, M., BOHME, H., DZIADZKA, M., KUNITZ, U., MAGNUS, R., AND VERWORNER, D. *Linux Kernal Internals.* Addison-Wesley, Reading, MA, USA, 1996.

[3] DEUTSCH, P., EMTAGE, A., BUNYIP, AND MARINE, A. How to use anonymous ftp. In *Request For Comments (RFC) : 1635* (October 1985).

[4] KLEIMAN, S. R. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Conference Proceedings* (Atlanta, GA, Summer 1986), USENIX, pp. 238–247.

[5] POSTEL, J., AND REYNOLDS, J. File transfer protocol. In *Request For Comments (RFC) : 959* (October 1985).

[6] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and implementation of the sun network filesystem. In *USENIX Conference Proceedings* (Portland, OR, Summer 1985), USENIX, pp. 119–130.

[7] STEVENS, W. R. *UNIX Network Programming.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1990.